LC3 Simulators | See execution, check results, debug code

--- See Machine's Content
Registers: R0-R7, IR, PC, MAR, MDR, PSR (in hex notation)
Memory: address/content (in hex, w/ translation to .asm)
Branch conditions: CC (usually as "Z" or "N" or "P")

--- Alter Machine's Content (except CC)
Registers
Memory location

--- Execute instructions:
STEP (execute 1 instruction)
RUN (execute w/o stopping)
STOP (stop execution)
BREAK (stop at location)

--- Set breakpoints:
Mark memory locations for BREAK

Most things work via double-clicking.
Breakpoint set: click a memory line or square icon.

[1.] Java (see projects/LC3-tools/PennSim.jar)

--- PennSim.jar: GUI, use double-click on items to change them. Hardware is slightly different from our LC3 and from PP's LC3. Don't use scroll bars, use up/down arrows on your keyboard.

[2.] Unix (see src/LC3-tools, see src/Makefile for compiling.)
--- LC3sim: commandline version of simulator
--- LC3sim-tk: X Windows GUI for LC3sim.

NB--The Makefile compiles the tools, then moves the executables to /bin.
I found that LC3sim and LC3sim-tk need to be moved back to src/LC3-tools for them to work. LC3-tools also has other executables (assembler and others) that we will need to use when we get to assembly language programming.

[3.] MS Windows (see src/PattPatel, or http://www.mhhe.com/patt2)

--- Simulate.exe: LC3 simulator w/ GUI
--- LC3edit.exe: (assembly language programming) editor and assembler.

LC3 OS services

Built in software, that is, pre-loaded into memory by simulator.
That would be a boot process in an actual machine.

TRAP	x25	Halt	--Halt: stop machine w/ message.
"	x20	Getc	--Getc: one char, keyboard ==> R0[7:0] (clears R0 first).
"	x21	Out	--Out: one char, R0[7:0] ==> display.
"	x22	Puts	--Puts: Mem[R0] ==> display (until x0000 found).
"	x23	In	--In: prompts, then one char input ala Getc.
"	x24	Putsp	--Putsp: Puts, but packed (2 chars per word).

A word on the difference between ascii representation of bits and actual bits. At a unix terminal window, enter

```
%> echo "abcd" | od -x1
```

You will see the ascii codes for each byte of input that "echo" sent to "od" (plus an extra byte for an assumed end-of-line). We would naturally think of this as the bytes of memory left-to-right. Now enter this,

```
%> echo "1234" | od -x1
```

You will again see ascii codes. The "real" bits for the first character, "1", are equivalent to x31, or 00110001 in actual bits. Next change the "x1" to "x2", and to "x4". You will see this,

31 32 33 34
3231 3433
34333231

If you think of the first byte in memory as containing the least-significant bits of a number, it would depend on the number of bytes the number had as to which byte you display first. If the number has 16 bits, then the first 16 bits would be expressed 3231 in hex, but if it was a 32-bit number, you would display 34333231 in hex. But, if we read things in right-to-left order, thought of memory as laid out right-to-left, and displayed bytes right-to-left, we would have,

"4321"
"4" "3" "2" "1"
34 33 32 31
3433 3231
34333231

In all cases, the least-significant bit is the rightmost, the least significant byte is the rightmost, and so forth. To accommodate the switching back and forth (and some other less important reasons), some machines put the most-significant byte of a number in the lowest byte address (called "big endian", versus "little endian").

Hex Notation

Base 16 (hexidecimal), positional notation for numbers:

(1) digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

(2) indicators: "0x", "x", "h" (indicates a hex representation)

d_{16} → a hex digit (also stands for value of digit)

$d_{16}^{(i)}$ → the i-th digit (")

hex representation		value
$d_{16}^{(3)}\ d_{16}^{(2)}\ d_{16}^{(1)}\ d_{16}^{(0)}$	⇒	$d^{(3)} \cdot 16^3 + d^{(2)} \cdot 16^2 + d^{(1)} \cdot 16^1 + d^{(0)} \cdot 16^0$
0x 1 2 3 4	⇒	$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0$

values of digits

hex	decimal	binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

hex	decimal	binary
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

hex-to-binary

x 1 2 3 4

simply replace hex digits w/ 4-bit equivalents in same left-to-right order.

$= (0001) \cdot 16^3 + (0010) \cdot 16^2 + (0011) \cdot 16^1 + (0100) \cdot 16^0$

$= (2^0) \cdot (2^4)^3 + (2^1) \cdot (2^4)^2 + (2^1 + 2^0) \cdot (2^4)^1 + (2^2) \cdot (2^4)^0$

$= 2^{12} + 2^9 + 2^5 + 2^4 + 2^2$

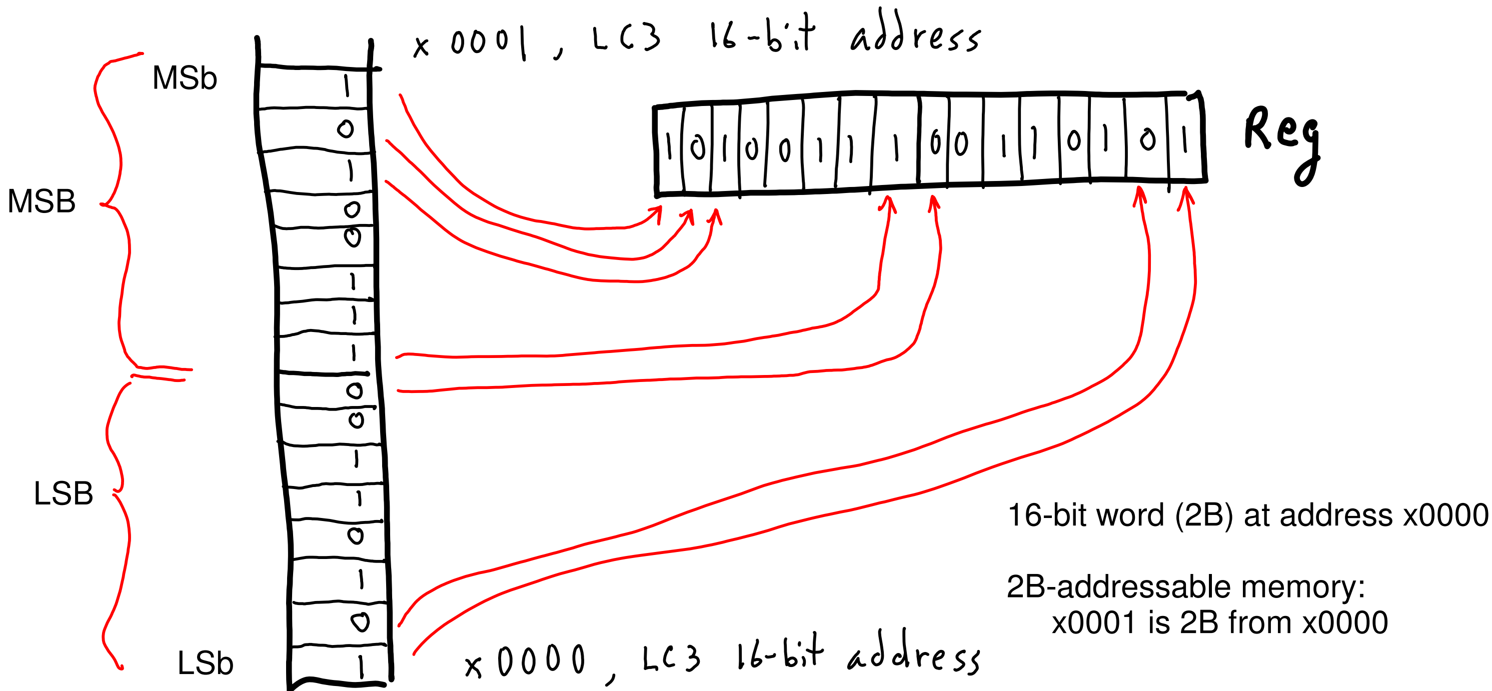
$=$ 0001 0010 0011 0100

$=$ 0001 0010 0011 0100 (in binary)

Last word on bit Layout in Mem, reg, ...

Memory (as bits)

LSb : least-significant bit
MSb : most-significant bit
LSB : least-significant byte
MSB : most-significant byte



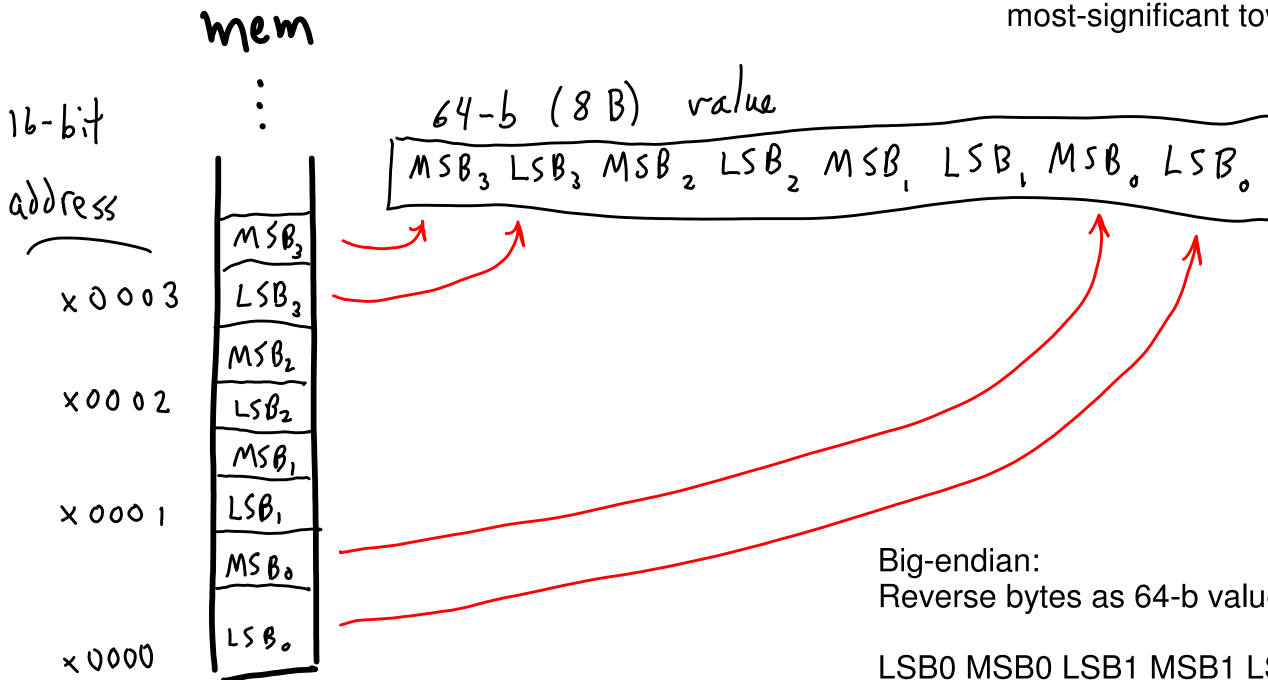
16-bit word (2B) at address x0000

2B-addressable memory:
x0001 is 2B from x0000

1B-addressable memory:
17-bit address 00000000000000000 ==> LSB
17-bit address 00000000000000001 ==> MSB

Small-end of memory: address x0000
Big-end of memory : address xFFFF

Little-endian:
least-significant toward small end
most-significant toward big end



Big-endian:
Reverse bytes as 64-b value:

LSB0 MSB0 LSB1 MSB1 LSB2 MSB2 ...